Hardware Realization of an FPGA Processor – Operating System Call Offload and Experiences

Hindborg, Andreas Erik; Schleuniger, Pascal; Jensen, Nicklas Bo; Karlsson, Sven

Published in:
Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing (DASIP)

Publication date:
2014

Hardware Realization of an FPGA Processor – Operating System Call Offload and Experiences

Andreas Erik Hindborg, Pascal Schleuniger
Nicklas Bo Jensen, Sven Karlsson
DTU Compute – Technical University of Denmark
{ahin,pass,nboa,svea}@dtu.dk

***Abstract*—Field-programmable gate arrays, FPGAs, are attractive implementation platforms for low-volume signal and image processing applications.**

The structure of FPGAs allows for an efficient implementation of parallel algorithms. Sequential algorithms, on the other hand, often perform better on a microprocessor. It is therefore convenient for many applications to employ a synthesizable microprocessor to execute sequential tasks and custom hardware structures to accelerate parallel sections of an algorithm. In this paper, we discuss the hardware realization of Tinuso-I, a small synthesizable processor core that can be integrated in many signal and data processing platforms on FPGAs. We also show how we allow the processor to use operating system services. For a set of SPLASH-2 and SPEC CPU2006 benchmarks we show a speedup of up to 64% over a similar Xilinx MicroBlaze implementation while using 27% to 35% fewer hardware resources.

I. Introduction

The ever increasing cost of developing a custom designed application specific integrated circuit, *ASIC*, has long since passed the point of feasibility for low volume embedded systems to include such custom components. Instead, designers look to FPGA devices for low volume markets, especially for signal and image processing. The performance and unit price of custom FPGA deigns are orders of magnitude lower and higher, respectively, than for custom designed ASICs. However, the attainable performance may be orders of magnitude higher than a solution that uses generic components.

Not all algorithms benefit equally from implementations on FPGAs. While it can be very efficient to implement parallel algorithms on FPGA devices, sequential algorithms are often better implemented on microprocessors. Therefore, signal processing applications often include microprocessors in the FPGA fabric. In this paper, we present our experiences in realizing a custom processor core that targets FPGA implementation. While we have previously used simulation to verify designs, in this paper we discuss how we realized the processor core, the *Tinuso-I* [1], on the Xilinx Zynq SoC platform. We propose and implement a method for offloading operating system services in an embedded system. The proposed system is composed of a hardware component and a software component. The hardware component provides communication interface logic while the software component provides a run-time library that allow client programs to use the communication interface logic.

We evaluate the system by executing selected SPEC CPU2006 and SPLASH-2 benchmarks. We demonstrate a speedup of up to 64% over a Xilinx MicroBlaze based baseline system.

To summarize, we make the following contributions:

- We show how we used the Xilinx Zynq SoC to realize the Tinuso-I processor core.
- We discuss the hardware bringup.
- We propose and implement a method for offloading operating system services and demonstrate it both with Tinuso-I and Xilinx MicroBlaze.
- We evaluate the performance of Tinuso-I compared to Xilinx MicroBlaze by executing SPEC CPU2006 and SPLASH-2 benchmarks.

The paper is organized as follows: In the next section the system architecture is described. Section III introduces the architecture, Tinuso, and its design philosophy. Section IV describes how we offloaded system calls, such as file accesses. The evaluation is discussed in section V whereas results are presented in section VI. Related work on synthesizable processor cores and system call handling is discussed in section VII and section VIII concludes the paper.

II. System Architecture

Several software layers and components are needed to execute real applications. Applications are commonly developed assuming a POSIX compliant operating system. However, running a full operating system on small embedded systems is often unfeasible.

We will use relatively large benchmarks for illustration. Such benchmarks are normally used to evaluate application processors such as mobile phone processors, tablet processors, desktop processors and server processors. Examples are the programs in the SPEC CPU2006 and SPLASH-2 suites.

To run these benchmark applications, certain POSIX operating system services must be available. For the SPEC CPU2006 programs, it is enough to provide the **open**, **close**, **read**, **write**, **stat** and **lseek** POSIX system calls. The SPLASH-2 applications also require the **gettimeofday** system call which is used to provide detailed timing metrics of the benchmark execution.

These services are usually provided by an operating system that runs locally on the processor core that requires the service. However, it is often not feasible to implement a new, or port an existing, operating system to resource constrained systems. To

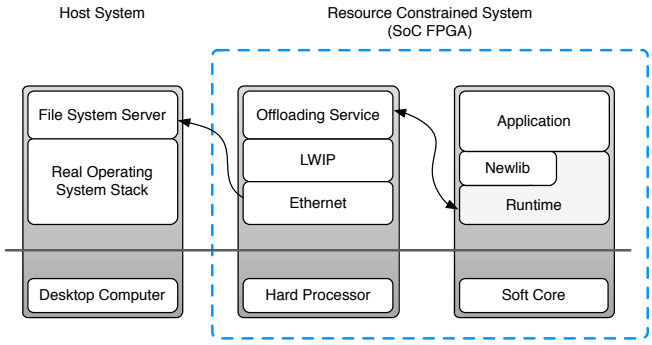

Fig. 1. Proposed architecture for running POSIX applications on resource constrained systems.

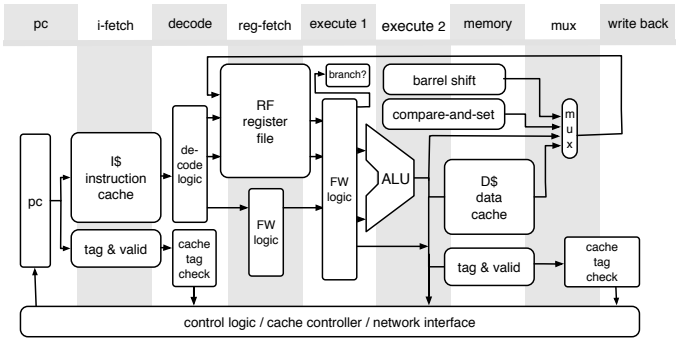

Fig. 2. The Tinuso-I pipeline architecture.

overcome this problem, we propose to offload operating system calls to a host system running a regular operating system. The architecture is presented in Fig. 1.

System-on-Chip FPGA devices are silicon devices that contain one or more application processors tightly coupled to an on-chip FPGA. In our system architecture, a Tinuso-I or Xilinx MicroBlaze system, as examples of resource constrained systems, are implemented in the FPGA fabric of the device. Requests for operating system services are relayed by a small runtime library to a service running on the application processor in the device. The application processor can then forward the request to a more capable device, or process the request locally if that is possible. In our experiments, and for convenience, we offload all requests to a desktop computer host.

We will later, in this paper, utilize the system architecture described in this section to evaluate the Tinuso-I processor with SPEC CPU2006 and SPLASH-2 benchmarks.

III. The Tinuso Architecture

Tinuso is a statically scheduled processor architecture optimized for high throughput when implemented on FPGAs [2]. Tinuso employs a single issue in-order pipeline with a load-store architecture [3]. Register file size, cache sizes, and cache organization are chosen to match the memory resources of current state-of-the-art FPGAs.

Tinuso aims to obtain a high instruction throughput by enabling super pipelined implementations. Tinuso leverages a hardware/software co-design approach where functionality that is not directly performance critical is moved from the hardware to the compilation toolchain. For example, the pipeline, with side-effects, is fully exposed to software. Thus, the compiler has to consider all types of hazards when scheduling instructions and must insert no-operation instructions, *nops*, if necessary. This approach gives the compiler sufficient control to eliminate many hazards, yielding a more lightweight hardware design that is easier to optimize and verify.

In processors with a large number of pipeline stages, branch instructions are resolved late in the pipeline which makes these instructions costly. Tinuso supports predicated instructions to reduce the number of dynamically executed branch instructions and to efficiently fill branch delay slots.

The instruction set architecture of Tinuso uses a fixed 32-bit word length with three operands. 7 bits are reserved for register operands, allowing for 128 general purpose registers. Tinuso supports 8 predication registers that can be used to leverage predicated execution.

Our current hardware implementation, the Tinuso-I core, is an instance of the Tinuso architecture. Tinuso-I applies super pipelining to obtain a high system clock frequency. Hence, register file, instruction cache, and data cache are implemented with pipelined block RAMs. These pipelined memory resources are found in state-of-the-art FPGAs [4]. Hence, each register file and cache access takes two clock cycles. To take full advantage of the fast memory accesses, Tinuso-I pipelines the execution stage into two stages, which results in a deep pipeline with 8 stages as depicted in Fig. 2.

Branch instructions in Tinuso-I are not resolved until the first execution stage. Thus, there are a total of 4 branch delay slots. Predicated instructions are supported to circumvent costly pipeline stalls due to branches. Predicated instructions also allow the compiler to transform if-else constructs into straight line code and thus avoid branch instructions. We have recent results indicating that compiler driven instruction scheduling using a modern compiler can be very efficient in using predicated instructions to avoid branch instructions and utilize branch delay slots.

The pipeline of Tinuso-I is fully exposed to the compiler. Tinuso-I has no pipeline interlocking logic, but supports forwarding from all pipeline stages where forwarding is possible without stalling. The pipelined memory access and the pipelined execution unit introduce a number of potential data dependency hazards. The compiler must resolve all these hazards when scheduling.

Cache misses are handled by invalidating instructions in the pipeline and restarting the pipeline when the cache miss has been resolved. This mechanism keeps the complexity of the control logic in the pipeline low, but makes cache misses more expensive as only one cache miss is resolved at a time.

The result of Tinusos design decisions is that we obtain an implementation that is both small and fast. By moving complexity away from the processor pipeline and into the compiler, the Tinuso-I implementation achieves a very high clock frequency, up to 376 MHz when synthesized to a Xilinx Virtex 6 device, and requires fewer FPGA resources than other commercial soft cores.

IV. Offloading Operating System Calls

In this section we present at method for offloading operating system calls for processor cores embedded in FPGA fabric. We focus our discussion on file system services, but the method is applicable to any operating system service.

A. File System Access

Providing simple file sytem access to a program running on a Tinuso-I core implemented in FPGA fabric amounts to providing the POSIX functions such as **open**, **close**, **read** and **write**.

In a regular computing system, this requires 1) access to a storage medium and 2) a software stack to provide a file system on top of the storage medium. One way to provide these services is to attach a storage controller to the processor, such as a SATA controller. The storage controller is used to provide access to an attached storage medium such as a hard drive. The storage controller needs to be managed by a driver running on the processor core. The file system services can then use the storage controller driver to back the file system service.

Since we are only interested in evaluating the performance of Tinuso-I, not in implementing a storage stack, we chose to leverage existing implementations of storage medium access and software stack, and only provide a thin shim to interface these existing services. We achieve this by intercepting file system service requests at the application level and offloading them to a regular desktop computer.

We intercept file system service requests by linking the benchmark applications with a custom run-time library. When an application requests a file system service, as when invoking the **open** system call, the run-time library sends the request to a desktop computer via a communication link. The response is received by the run-time and relayed back to the requesting program. With this approach it is possible to provide file system access to a processor core by only implementing the following:

- A minimal run-time library that intercepts file system service requests from programs running on the Tinuso-I core in FPGA fabric.
- Communication link support on the desktop computer that services the file system requests and on the soft core.
- A service on the desktop computer that responds to the file system service requests.

With this approach it is possible to leverage the entire storage stack available on a regular desktop computer, without porting the entire file system stack to the system under test.

B. Implementation

We implement the proposed offloading method for a Tinuso-I core synthesized to the FPGA fabric of a Xilinx XC7Z020-CLG484-1 Zynq device [5]. The silicon device is part of an AvNet ZedBoard development kit [6]. The Zynq device is a complex unit with many components, including two ARM Cortex A9 cores, an 1G Ethernet MAC and an FPGA fabric area. We use TCP/IP over the 1G Ethernet MAC to provide the communication link between Tinuso-I and the desktop computer.

We compile applications for the Tinuso-I using an embedded binutils/GCC/Newlib tool chain [7], [8], [9]. A Standard C library API is provided by the Newlib C library. Newlib services file system requests by calling user supplied methods that implements the services.

A visual representation of the system architecture is given in Fig.3. The Tinuso-I core is placed in the FPGA fabric. It has an interrupt line capable of interrupting the Cortex A9 cores, a 64 bit wide burst capable AXI Master [10] interface connected to the on chip DDR memory controller and a 32 bit wide AXI Lite [10] slave interface connected to the Cortex A9 cores for configuration. We use one of the Cortex A9 cores to bootstrap the system and load the Tinuso program into main memory. In order not to port the driver for the 1G Ethernet MAC to Tinuso, we run the driver and the TCP/IP stack on the ARM core.

When Tinuso application calls the **open**, **close**, **read** or **write** C library functions, a request structure is populated by the runtime. The structure is stored in main memory. An interrupt is raised by the runtime on the Tinuso core to notify the ARM core of the pending request.

The ARM core sends the request via 1G Ethernet to the desktop computer. The desktop computer services the request using a local disk and responds with data if applicable. The ARM core stores this data in main memory and notifies the Tinuso-I core that the request has been serviced by writing a hardware register in the Tinuso-I core. This causes the Tinuso-I core to lower the interrupt and continue application execution. The steps for servicing a file system call, as depicted in Fig.3, are:

1) Tinuso-I populates a request structure and installs a pointer to the structure on a predifined memory location.
2) Tinuso-I core raises a level sensitive interrupt flag for the ARM core.
3) The ARM core reads the request structure and dispatches the request to the desktop computer host via Ethernet.
4) The desktop computer host executes the service request locally and returns the result.
5) The ARM core places the result delivered by the host desktop computer in the buffers used by the Tinuso-I core.

V. Evaluation

We evaluate the hardware system described in section IV-B by running a number of benchmarks that is widely used to evaluate the performance of computing systems. We run a selection of benchmarks from the SPLASH-2 [11] and SPEC CPU2006 [12] benchmark suites. The benchmarks include several scientific computations, artificial intelligence algorithms and algorithms used in computer graphics. We scale the workloads to achieve execution times between 1 and 15 minutes. The benchmark names and workload sizes are presented in table I.

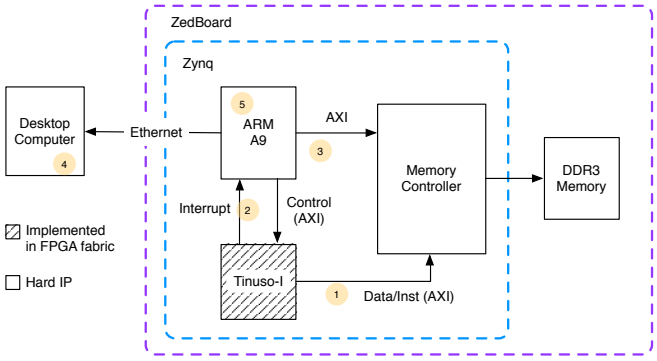

Fig. 3. Test system architecture overview. A single Tinuso-I core is instantiated in the FPGA fabric of a Xilinx Zynq device. File system access is provided by a desktop computer connected by Ethernet.

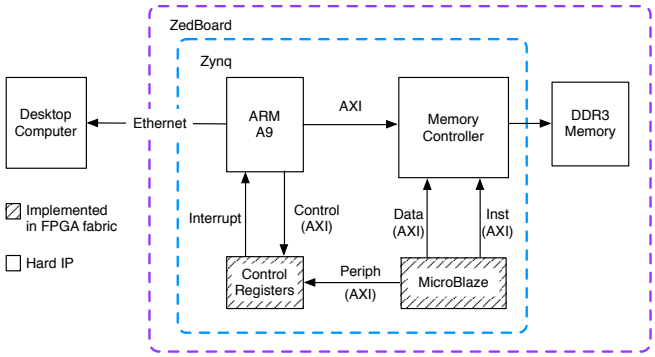

Fig. 4. Test sytem architecture for the MicroBlaze baseline system.

TABLE I. WORKLOAD PARAMETERS FOR THE BENCHMARKS.

Benchmark	Workload Size
445.gobmk	capture.tst
458.sjeng	Defaut test input
462.libquantum	Default training input
fmm	2048 elements
ocean-noncont	258x258 grid points
radiosity	large-room
raytrace	teapot
water-nsquared	512 molecules
water-spatial	512 molecules

445.gobmk is an artificial intelligence program that plays the GO game. The program repeatedly reads game positions and analyses the game for the next move. **458.sjeng** is an artificial intelligence program that plays chess. The program takes as input a chess position and performs a game tree search to find a good move. **462.libquantum** is a quantum computing simulator. It simulates a quantum computer algorithm that factorizes integers in polynomial time. All of the programs from the SPEC CPU2006 suite are integer programs that contain no floating point calculations.

fmm is an implementation of the Fast Multipole Method for simulation of the N-body problem. **ocean** is a large scale simulation of ocean movement and eddy currents. **radiosity** is program that calculates the equilibrium distribution of light in a scene. **raytrace** is a program that renders 3D graphics with a ray tracing algorithm. **water** is a physical N-body simulation of water molecules. All of the programs from the SPLASH-2 suite are floating point programs. The SPLASH-2 programs are designed to be executed in multi-processor systems, but have been modified to execute only one thread for the use in this paper.

A. Tool Chain

We compile the benchmark programs for Tinuso-I using our own GCC 4.9.0 based tool chain. The benchmarks are linked with Newlib version 1.20.0. All benchmarks and the C library are compiled with optimization level **-O2**.

B. Execution Time Measurements

For the SPLASH-2 programs we use the built-in timing measurements to report execution time. This logic depends on the **gettimeofday** system call. This system call is handled at the connected desktop computer in the same way that file system calls are handled. The SPEC programs do not include timing logic. We measure the execution time of the SPEC programs by starting a timer on the ARM core when the Tinuso-I core is released from reset. The timer is sampled again when the the exit system call is invoked by the program running on the Tinuso-I core.

C. Baseline Platform

To provide a baseline, we implement a MicroBlaze based system that is as similar as possible to our Tinuso-I based system, see Fig.4 for reference. Due to architectural differences between MicroBlaze and Tinuso-I, the topology of the systems cannot be completely identical. In the Tinuso system, we use a set of system registers embedded inside Tinuso-I to control the Tinuso-I core. Because there is no such registers in MicroBlaze, we use an AXI slave peripheral containing two registers for this purpose in the MicroBlaze system. The registers are used by the MicroBlaze core to raise the Cortex A9 interrupt line, and by the Cortex A9 core to acknowledge interrupts from the MicroBlaze core.

The MicroBlaze IP core requires three AXI interface connections. A non-cached AXI-Lite Master interface is used to communicate with the control register bank. Two separate burst capable AXI Master interface is used to service the instruction and data caches. The latter two interfaces are connected to separate AXI slave ports on the Zynq memory controller.

We confgure the MicroBlaze IP core similarly to our Tinuso-I core, with separate 16 kilo byte direct mapped instruction and data caches using 32 byte cache lines. We enable the barrel shifter, disable the multipier, the divider and the floating point unit.

We compile the benchmarks for the MicroBlaze system using GCC compiled from the official Xilinx GCC 4.8 branch [13]. The binaries are linked with Newlib 1.19 compiled from Xilinx sources [14].

D. Performance Counters

We characterize the memory system for each of the benchmarks by using hardware profiling counters in the Tinuso-I core. The profiling counters count the following:

- Instruction cache misses

- Data cache misses
- Data cache write-back events
- Cycles spent waiting for instruction cache
- Cycles spent waiting for data cache
- Cycles spent on write-back operations by the cache controller

Enabling the profiling counters in the Tinuso-I core lowers the maximal operating frequency of the core. Therefore we obtain the profiling information separately from the execution time information.

E. Hardware Configuration

The desktop computer host used in the experiments is a 2013 MacBook Pro with a 2.4 GHz Intel i5-4258U CPU and 8 GB of 1600 MHz DDR3 memory.

The ZedBoard is configured with 512 MB of DD3 memory on a 32 bit bus operating at 1066 MT/s. The board is equipped with the Xilinx XC7Z020-CLG484-1 System-on-Chip device.

The ARM core in System-on-Chip is clocked at 667 MHz and is attached to separate 4 kilo byte, 4-way set associative L1 instruction and data caches. The L1 caches interface a shared L2 cache. The L1 caches are non-blocking with support for 4 outstanding reads and 4 outstanding writes, and speculative pre-fetching. The L2 cache is a 512 kilo byte 8-way set associative cache. All caches use write-back policy and 32 byte lines. The A9 core has a 4 slot 64 bit store buffer with data merging support.

Tinuso-I is configured with separate 16 kilo byte direct mapped instruction and data caches with a line size of 32 bytes. The data cache uses write-back policy. The cache controller is capable of handling one outstanding miss at a time. On a data cache write-back, dirty data will be written back before a read of the new data is issued.

For the parts of the system placed in the FPGA fabric, the full FPGA flow including synthesis, mapping, placement and routing is performed in Xilinx's most recent Vivado 2013.4 Design Suite.

VI. RESULTS

The hardware implementation results for both the Tinuso and MicroBlaze based systems are based on the "Place and Route" report of the Xilinx Vivado Design Suite. Table III lists the MicroBlaze settings in the IP configurator. Parameters are configured to match the Tinuso-I pipeline and its memory hierarchy.

Table IV shows the FPGA resources required by the Tinuso based system compared to those required by the MicroBlaze based system. The Tinuso system utilizes about 27% fewer registers and 35% fewer LUTS than a similar MicroBlaze based system.

TABLE II. MAXIMUM CLOCK FREQUENCY FOR THE MICROBLAZE SYSTEM AND THE TINUSO-I SYSTEM.

System	Clock Frequency (MHz)
MicroBlaze	115
Tinuso-I	168

TABLE III. MICROBLAZE CONFIGURATION.

Pipeline configuration	
Pipeline stages	5
Branch Target Cache	enabled
Branch Target Cache Size	default
Barrel Shifter	enabled
Integer Multiplier	disabled
Integer Divider	disabled
Floating Point Unit	disabled
Cache configuration	
Instruction and Data Cache	enabled
Size in Bytes	16kByte
Line length (words)	8
Enable Write-back Storage Policy	enabled
Number of Victims	0

TABLE IV. FPGA RESOURCE UTILIZATION THE MICROBLAZE SYSTEM AND THE TINUSO-I SYSTEM.

Resource	MicroBlaze System	Tinuso-I System	Reduction
LUTs	2946	2143	27%
Registers	2811	1811	35%
F7 Muxes	128	31	75%
F8 Muxes	2	0	100%

Even though Tinuso-I employs a deeper instruction pipeline than MicroBlaze it utilizes fewer hardware resources. The low hardware resource usage of Tinuso-I is a result of the design choice to move complexity from hardware to software, which enables a simpler pipeline design. The Tinuso-I compilation toolchain considers all types of hazards, therefore there is no need for dependency checks and interlock logic in hardware. Moreover, the Tinuso instruction set architecture is simpler than the MicroBlaze instruction set architecture. For example, Tinuso-I only implements memory operations for 32-bit data types. Thus, memory accesses of shorter data types in Tinuso-I require mask and shift instructions.

Table II lists the maximum system clock frequency for both the Tinuso based system and MicroBlaze based system. While the Microblaze system can be clocked at 115 MHz, we achieve timing closure for the Tinuso based system at 168 MHz. The higher clock frequency of the Tinuso based system can be attributed to the pipelined cache memories, the pipelined register file, the pipelined execution stages, and the exposed pipeline architecture.

We identified the time critical path of the design of the Tinuso-I configuration as the routing delay of the interface to the AXI memory controller. It is a possibility to add more pipeline stages in the AXI interface logic to obtain a higher clock frequency and better routing options. However, this will add more clock cycles latency for each cache miss. Hence, the performance of the Tinuso-I core is limited by the implemented memory hierarchy.

A. Code Execution Time

The execution time for the benchmarks are depicted in Fig.5. The figure shows the execution time for the benchmarks when executed on a 168 MHz Tinuso-I system and a 115 MHz MicroBlaze system, normalized to the latter. The results show that the Tinuso-I system achieves a speedup of up to 64%. We run each benchmark 15 times on each platform and compute the average runtime. The runs are very consistent with less

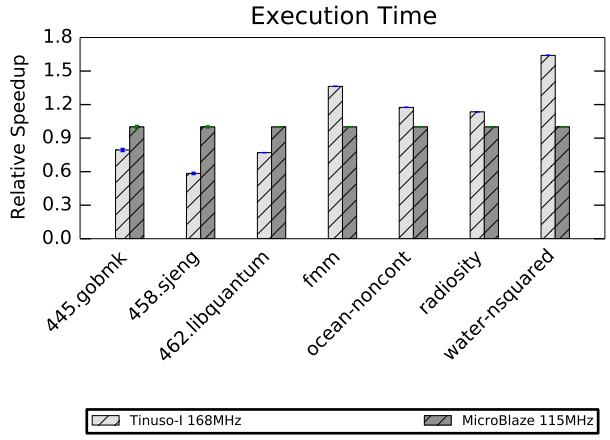

Fig. 5. Execution time of the benchmarks normalized to the execution time on the MicroBlaze system.

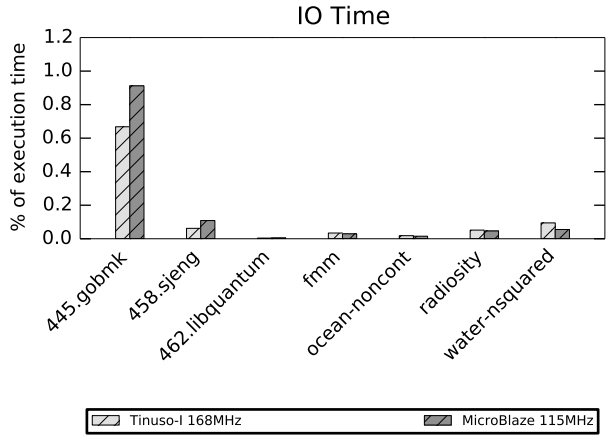

Fig. 6. Average fraction of execution time spent in the IO system.

than 1% difference between each run of the same benchmark on the same platform.

Fig.6 shows the average time spent in the IO system. The IO time is just a tiny fraction of the entire execution time, no larger than 1% and in most cases less than 0.2%.

Tinuso-I performs significantly better than MicroBlaze on benchmarks that execute many floating point operations. The compiler successfully leverages predicated instructions of Tinuso to produce very efficient software based floating point algorithms.

The Tinuso-I system is slower than the MicroBlaze system on the three SPEC benchmarks. To better understand these results, we have implemented profiling counters in the cache controller of the Tinuso-I system. The profiling counters measure the number of cache misses and the number of cycles the system spend waiting for the cache controller. Fig.7 presents the number of misses in the instruction cache divided by the execution time of the program. Fig.8 shows the same

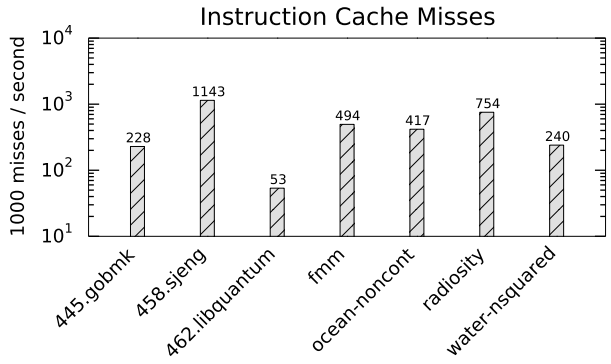

Fig. 7. Number of instruction cache misses normalized to execution time.

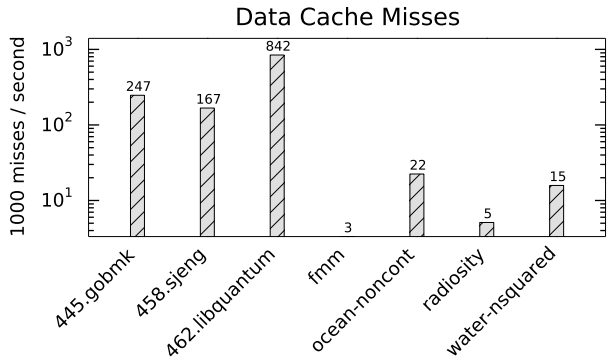

Fig. 8. Number of data cache misses normalized to execution time.

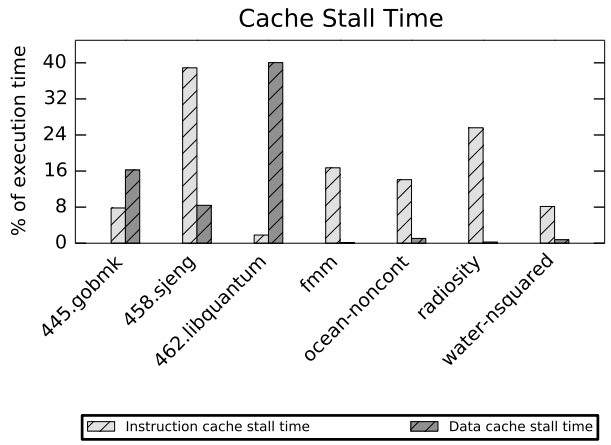

Fig. 9. Time spent waiting for cache.

metric, but for the data cache. For the three SPEC benchmarks where Tinuso-I is slower than MicroBlaze, the number of data cache misses is significantly higher. Fig.9 shows the fraction of the execution time that is spent waiting for the caches by the Tinuso-I system. For **458.sjeng** and **462.libquantum**, execution time is heavily influenced by cache stalls.

Tinuso-I uses 16 kilo bytes of one way direct mapped cache for instruction and data each. These caches are interfaced to a blocking cache controller that is implemented in the hard silicon DDR memory controller available via the Zynq AXI interconnect. Tinuso-I does not support prefetching or speculative execution, so every cycle spent waiting for memory is effectively wasted time. The latency of memory operations therefore has a very high impact on the performance of the Tinuso-I system. Based on the performance counters in the cache controller, we determine the average wait time for a data cache access is 50 cycles. It is clear that if memory subsystem performance can be improved, the system performance will benefit significantly.

445.gobmk includes a high number byte sized memory operations, as it uses bytes to represent game state. Tinuso-I only supports aligned full-word memory operations. To access a short data types in main memory, Tinuso-I must perform a memory operation and apply shift and mask operations to extract the data. Due to scheduling constraints on memory and shift operations, there is an overhead to accessing short data types in main memory.

VII. Related Work

Major FPGA vendors such as Xilinx, Altera and Lattice Semiconductors offer synthesizable processor cores optimized for their respective technologies. These cores are highly configurable and come with a large number of peripherals and rich tool-chain support. Xilinx MicroBlaze and Altera Nios II come as optimized netlists of vendor specific primitives. Hence, they are bound to the vendor's hardware and toolchains.

Xilinx's MicroBlaze is a popular synthesizable processor that implements a 32-bit Harvard RISC architecture with a rich instruction set optimized for embedded applications [15]. The performance optimized MicroBlaze configuration utilizes a five-stage pipeline. A large amount of peripheral, memory and interface features are available to adapt the processor to a given application. MicroBlaze comes with an AXI interface which can be used to connect to memory controllers or to build multicore systems.

Altera's current equivalent to MicroBlaze is called Nios II [16]. The Nios II family includes three processors that are optimized for highest performance, smallest size, and performance and size balanced implementation. NIOS II allow for up to 256 custom instructions which can be used to tune the system to improve the performance of dedicated signal and image processing applications.

There exists a plethora open source synthesizable processor cores, such as LEON3 and OpenRISC 1200. However, the available instances of these processor cores are not optimized for any specific target technology and can therefore only be clocked at a relatively low clock frequency. As a result, performance and instruction throughput are low when implemented on an FPGA [17].

The LatticeMico32 is another open-source processor design provided by Lattice Semiconductors [18]. It is available in synthesizable register transfer language and can be ported to any FPGA family. LatticMico32 is an in-order single issue processor with a load-store instruction set. Although LatticeMico32 comes with a 6-stage pipeline, the performance is typically lower than a MicroBlaze configuration. One reason for this is the high branch cost in LatticeMico32 caused by the static branch prediction. LatticeMico32 comes with a Wishbone-interface for connection to memory controllers.

SCOORE is an attempt to implement an of out-of-order processor on FPGA devices [19]. However, efficient implementation of out-of-order architectures require the use of fully associative memories, which are not efficiently implemented in FPGA fabric.

Multiple projects describe methods that allow soft processor cores, programmable accelerators, or hardware threads to access operating system services in way that is transparant to the requesting entity. In ReconOS, the concept of hardware threads is introduced [20]. A hardware thread in ReconOS is an HDL core that is loaded into FPGA fabric. The HDL core interacts with the operating system through a predefined interface and a state machine that implements a predefined protocol. In ReconOS terms, a Tinuso-I core could be considered a hardware thread. ReconOS has been extended with transparent address translation in the ReconOS VM system [21].

Khiar et al. describe the Flexible Operating System for Reconfigurable Hardware, FOSFOR [22]. Like ReconOS, FOSFOR provides an abstraction for hardware components and processors in re-configurable fabric that allow them to communicate and access operating system services.

Predicated execution is a well know architectural feature [23]. It is often used in SIMD engines such as graphics processors to allow efficient software pipelinening of loops [24]. Mahlke et al. show that the use of hyperblock scheduling on a fully predicated instruction set results in an average speedup of 72% on an 8 issue processor [25] Allen et al. have demonstrated how control dependence in the instruction stream can be translated into data dependence using an if-conversion pass in the compiler back end [26].

Tinuso-I has been demonstrated in high performance signal processing applications such as Synthetic Aperature Radar applications [27] and Microwave Imaging [28].

VIII. Conclusions

In this paper, we have shown how we used the Xilinx Zynq SoC to realize the Tinuso-I processor core and to perform a hardware bringup. We have proposed a method for offloading operating system services using the ARM host of the Xilinx Zynq SoC. This proposed system for offloading operating system services is highly relevant for prototyping and simulating signal and data processing applications. For example, it allows for emulating a camera system by operating on a set of files.

We have demonstrated our method by using it to evaluate both Tinuso-I and Xilinx MicroBlaze. We evaluate the system by executing a set of SPEC CPU2006 and SPLASH-2 benchmarks. We measure a speedup of up to 64% for Tinuso-I

over a similar Xilinx MicroBlaze baseline system. The Tinuso-I system uses 27% fewer LUTs and 35% fewer registers than the MicroBlaze system. Tinuso-I is highly configurable and the simple architecture allows for easy extension with dedicated hardware blocks. Tinuso therefore is an attractive platform for a broad range of embedded system signal and data processing applications.

ACKNOWLEDGMENTS

This work has been partially funded by the ECSEL JU as part of the COPCAMS project under GA number 332913.

REFERENCES

[1] P. Schleuniger, S. McKee, and S. Karlsson, "Design principles for synthesizable processor cores," in *Architecture of Computing Systems*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7179, pp. 111–122.

[2] P. Schleuniger, "Performance aspects of syntheizable computing systems," Ph.D. dissertation, Technical University of Denmark, 2014.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The hardware/software interface*, 4th ed. Morgan Kaufmann, 2008.

[4] Xilinx, *7 Series FPGAs Memory Resources User Guide (UG473)*, v1.7 ed., October 2012.

[5] ——, *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, 1st ed., March 2013.

[6] AVNET, *ZedBoard Hardware User's Guide*, 1st ed., Jan 2013.

[7] GNU Binutils Developers, *GNU Binutils*, http://www.gnu.org/software/binutils/, retrieved on May-18-2014.

[8] R. Stallman and the GCC Developer Community, "GNU compiler collection internals, for gcc version 4.9.0," http://gcc.gnu.org/onlinedocs/gcc-4.9.0/gccint.pdf, 2013, retrieved on May-18-2014.

[9] Newlib Developers, *Newlib*, https://sourceware.org/newlib/, retrieved on May-18-2014.

[10] ARM, *AMBA AXI and ACE Protocol Specification*, October 2011.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ACM, 1995, pp. 24–36.

[12] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[13] Xilinx, "Xilinx GCC Repository at Github," https://github.com/Xilinx/gcc, 2014, retrieved on May-05-2014.

[14] ——, "Xilinx Newlib Repository at Github," https://github.com/Xilinx/newlib, 2014, retrieved on May-05-2014.

[15] ——, "MicroBlaze Processor Reference Guide UG081 v12.0," www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf, 2011, retrieved on October-5-2013.

[16] Altera, "Nios II Processor Reference Handbook NII5V1-11.0," www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf, 2011, retrieved on October-7-2013.

[17] J. Tong, I.Anderson, and M. Khalid, "Soft-core processors for embedded systems," in *Proceedings of the 18th International Conference on Microelectronics*. IEEE, 2006, pp. 170–173.

[18] Lattice Semiconductor, "Latticemico32 processor reference manual v8.1," www.latticesemi.com/documents/lm32_archman.pdf, 2010, retrieved on March-25-2013.

[19] F. J. Mesa-Martinez *et al.*, "SCOORE: Santa Cruz Out-of-Order RISC engine, FPGA design issues," in *Workshop on Architectural Research Prototyping (WARP), held in conjunction with ISCA-33*, 2006, pp. 61–70.

[20] E. Luebbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 1–33, 2009.

[21] A. Agne, M. Platzner, and E. Lubbers, "Memory virtualization for multithreaded reconfigurable hardware," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 185–188.

[22] A. Khiar, N. Knecht, L. Gantel, S. Lkad, and B. Miramond, "Middleware based executive for embedded reconfigurable platforms," in *Design and Architectures for Signal and Image Processing*. IEEE, 2012, pp. 1–6.

[23] J. C. Park and M. Schlansker, "On predicated execution," Technical Report HPL-91-58, HP Labs, Tech. Rep., 1991.

[24] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, no. 1. Wiley Online Library, 2007, pp. 80–113.

[25] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*. IEEE, 1992, pp. 45–54.

[26] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th Symposium on Principles of Programming Languages (POPL)*. ACM, 1983, pp. 177–189.

[27] P. Schleuniger, A. Kusk, J. Dall, and S. Karlsson, "Synthetic aperture radar data processing on an FPGA multi-core system," in *Architecture of Computing Systems*. Springer, 2013, pp. 74–85.

[28] P. Schleuniger and S. Karlsson, "A synthesizable multicore platform for microwave imaging," in *10th International Symposium on Applied Reconfigurable Computing*. Springer, 2014, pp. 197–204.